A scientist's guide to cloud computing

Editors: Konrad Hinsen, konrad.hinsen@cnrs-orleans.fr | Matthew Turk, matthewturk@gmail.com

A Scientist's Guide to Cloud Computing

Sotirios A. Tsaftaris IMT Institute for Advanced Studies Lucca, Italy, and Northwestern University

New tools (some commercial and even public), have made it so that dealing with the cloud and running large-scale processing can be rather easy and efficient.

My PhD work in 2003 entailed the design of 256 DNA sequences of length 45 capable of encoding 256 integers, and satisfying a number of constraints. Understandably, the search space is vast (since there are four DNA bases, it is 4^{45}), and unfortunately being combinatorial, this problem is known to be NP-hard. My best algorithm (as most algorithms for DNA codeword design) used stochastic local search (similar to genetic algorithms), which might still converge to one of the many local minima; thus, the traditional approach is to run many instances of it with different start conditions and parameters to obtain the best solution. This is a pure example of simple-to-parallelize problems—many instances that can be executed independent from each other. Numerous scientific problems fall in the category of embarrassingly parallel: for example, Monte Carlo simulations, parameter and threshold sensitivity curves (where a span of parameters are explored and the performance of the algorithm is evaluated), and obtaining rate distortion curves in compression (where for different bitrates we must obtain the distortion).

A multiprocessor machine in my former lab could handle the execution of many instances. However, many colleagues needed access to the same machine at the same time, particularly around conference deadlines. Relying on our university's cluster was a solution in the long run, but it required a proposal explaining the need for, and amount of, compute hours, approved by a university committee, and budgeted (after a certain quota) to an account. Once approved, it entailed convincing the cluster administrator to install software not present on the cluster, and then depending on the priority as a user, waiting an unknown amount of time for jobs to start and finish (at the control of our cluster's Portable Batch System [PBS] scheduler). My approach to satisfy on-the-spot needs was simpler: I wrote a master-slave framework to send jobs to Matlab "slaves" via a shared directory in our fileserver (this was done prior to Matlab's distributed computing server offering). I was using all the desktop PCs of the lab, which had different capabilities, to solve the combinatorial problem. This helped tremendously in speeding up my computation.

In 2006, Amazon introduced Amazon Web Services (AWS; http://aws.amazon.com), which included the Elastic Compute Cloud (EC2; http://aws.amazon.com/ec2) service. Not many people in the scientific world (except the expert community in grid, utility, and high-performance computing) realized the impact at the time. For me, the benefit was clear and immediate. For example, when I had to invert a large matrix for an image restoration project, I signed up with EC2 and picked an instance satisfying my needs. I then launched a virtual machine (a software-based implementation resembling a physical computer), installed the needed application, and solved the problem without much hassle and logistics overhead. However, coordinating many instances in a manner that was straightforward and transparent for me, the user, was another story.

Later, after I had moved to a new institution, I ran into an issue with large-scale processing. We were working in my lab with neuroimaging data and multiple processes required several hours. Navigating the system again towards finding the cluster infrastructure was too slow. I searched neuroimaging and the cloud and found discussions in forums describing procedures using AWS to execute neuroimaging pipelines (which are now better documented for Freesurfer at http://freesurfer.net/fswiki/AmazonCloud, and for others at www.nitrc.org/plugins/mwiki/index.php/nitrc:User_Guide_-_NITRC_Computational_Environment), but not at a large, coordinated scale. I searched for cloud computing tools and a few options came up.

I believe my story will resonate with many readers. Clearly, the advent of commercial clouds has revolutionized everyday computing, but many now believe it will impact scientific computing, as well. Already several articles have been written illustrating and evaluating the potential of cloud computing for science, particularly from the software engineering and resource management side (see the related sidebar for more information). This article is geared instead towards helping readers understand that with some new (commercial and even public) tools, dealing with the cloud and running large-scale processing can be rather easy and efficient.

Cloud Computing Solutions for the Scientist

Several commercial entities, academic groups, and individuals considered the problem of how to facilitate interaction with cloud infrastructures from the scientists' perspective. For example, *Cycle Computing* (www.cyclecomputing.com)

 1521-9615/14/$31.00 © 2014 IEEE Copublished by the IEEE CS and the AIP

Cloud Computing for Science from the View of CiSE

Cloud's appeal for science is clear: simplicity, elasticity (that is, the availability of large resources on the spot by launching as many instances as needed), true reproducibility (the virtual machine and the code running on that machine can be made public together with the data, when necessary), the ability to cover a large span of open questions previously unattainable due to a possible lack of computing power, and most importantly, democratization of science (since anyone has access to large computing power).

Naturally, this journal has been following scientific cloud computing closely, with two special issues[1,2] in 2009 and 2013 and articles throughout the years.[3] In the most recent special issue, review,[4] comparison,[5] and application[6-7] articles appeared, covering the cloud's broad potential as well as some of its challenges.

References

1. F. Sullivan, "Cloud Computing for the Sciences," *Computing in Science & Eng.*, vol. 11, no. 4, 2009, pp. 10–11.
2. G.K. Thiruvathukal, and M. Parashar, "Cloud Computing," *Computing in Science & Eng.*, vol. 15, no. 4, 2013, pp. 8–9. 2013.
3. J.J Rehr et al., "Scientific Computing in the Cloud," *Computing in Science & Eng.*, vol. 12, no. 3, 2010, pp. 34–43.
4. M. Parashar et al., "Cloud Paradigms and Practices for Computational and Data-Enabled Science and Engineering," *Computing in Science & Eng.*, vol. 15, no. 4, 2013, pp. 10–18.
5. G. Juve et al., "Comparing FutureGrid, Amazon EC2, and Open Science Grid for Scientific Workflows," *Computing in Science & Eng.*, vol. 15, no. 4, 2013, pp. 20–29.
6. D.K. Krishnappa et al., "CloudCast: Cloud Computing for Short-Term Weather Forecasts," *Computing in Science & Eng.*, vol. 15, no. 4, 2013, pp. 30–37.
7. Y. Simmhan et al., "Cloud-Based Software Platform for Big Data Analytics in Smart Grids," *Computing in Science & Eng.*, vol. 15, no. 4, 2013, pp. 38–47.

offers their consulting expertise to users and builds cloud-based high-performance computing (HPC) alternatives; however, they don't offer a tool or a service for a user to develop their own cloud-based infrastructure.

StarCluster (http://star.mit.edu/cluster), on the other hand, is an open source toolset for simplifying and automating the creation and management of cloud-based clusters geared for science with AWS. The StarCluster Amazon Machine Image (AMI is the virtual machine image of AWS) includes several options, packages, and plugins preinstalled, ready to work in a StarCluster arrangement. It can also be extended with user-defined software/applications and plug-ins. For example, with such a plug-in (see http://star.mit.edu/cluster/docs/latest/plugins/ipython.html?highlight=ipython), an interactive IPython cluster configuration is even possible (ipcluster enables the use of multiple computational instances within the Python interpreter). StarCluster provides several job-handling options (such as Sun Grid Engine), and Hadoop and OpenMPI implementations. StarCluster is definitely a powerful open source tool for scientific computing in the cloud; however, an expertise and comfort on cluster setup from the user is required. Since not all scientists are well-versed, for example, with cluster configuration and management and job queues, several approaches towards simplifying access to such computational power are under development.

An interesting academic and open source attempt towards this goal is *OpenCPU* (https://public.opencpu.org), which focuses on calling and using the R language (www.r-project.org) on a cloud server via a Web API. Using OpenCPU is easy and when combined with JavaScript, R functions and scripts can be embedded in the browser, and user-defined apps (R packages bundled with Web content for headless operation) can be deployed and shared. Interested users deploy OpenCPU on EC2 or other compatible providers and pay solely for the cost of the cloud provider. However, currently, OpenCPU doesn't offer an architecture where multiple cloud instances can be used to tackle the same problem in an efficient fashion.

A recent start-up *CPUsage* (www.cpusage.com), released its public beta in September 2013 with the goal of facilitating submitting and distributing tasks in the cloud, with the focus on being application, cloud provider, and language agnostic. The user can manage and submit jobs via a Representational State Transfer (REST) API or command-line tools (which will be released soon). The user customizes a Linux Container with their own applications and software, running on Amazon AWS, which is then wrapped in an API. Containers aren't technically independent operating systems, but appear as such to the end user. Current planned enhancements include API and application builders, and libraries for known languages such as Python. CPUsage also intends to benchmark a user's jobs and advise upon what's the best provider and instance configuration, either optimizing cost or speed/efficiency. The pricing strategy follows a per-minute compute cost for various configurations with access to a multinode infrastructure. Data transfer costs are included within the price, while no storage is available within CPUsage, except the 500 Mbytes of temporary storage within the container.

Relying on Python's emergence and scientific appeal,[1] *Wakari* (www.wakari.io; by Continuum Analytics) deploys

a "Python in the cloud" system. Thus, the user has access to a Python instance on Wakari's cloud environment, interactive even within a Web-based IPython notebook. The computational power depends on the choice of instance, with multicore, high-memory, and ipcluster options available. The user can also customize a Linux Container to include other applications/libraries. Using Wakari requires a monthly subscription, which varies in cost according to the chosen setup. Overall, for those familiar with Python, it's really easy to use the Web-based notebook with Wakari, and this appeals to users who want to create and promote reproducible research; however, while adding extra nodes is possible, it's less intuitive.

PiCloud (www.multyvac.com) is one of the first commercial entities with a special focus on making scientific computing in the cloud simple for the users. PiCloud provisions AWS instances transparently to the user, acting as a middleware between AWS and the user. Their provisioning technology allows PiCloud to compete for the lower cost spot instances on Amazon. By predicting clients' usage, they bid for future AWS instances as needed. They bundle their own instances (referred to as *core types*) so that many users can co-exist within a single (for example, large) AWS instance. This allows PiCloud to offer competitive compute pricing at the millisecond. Users pay for data storage and data download. In addition, PiCloud offers several instance configurations and even permits a user to define multiple instances for a single job, in case high memory or multicore processing is needed. PiCloud is appealing due to the amount of optimization involved behind the scenes to simplify access to the cloud. The user interacts transparently with the cloud via an API and a Python library.

A "Getting Started" Tutorial with PiCloud

Here, I'll demonstrate with two examples the power of PiCloud for distributed processing. In the following, a fixed-width font is used for Python and PiCloud commands and functions. At the time of writing, PiCloud offered 20 hours of free computing, so readers can easily test PiCloud following the examples below.

The first step is to create a PiCloud account and login. The Web interface of PiCloud provides several options, such as reviewing of billing, and managing data storage and environments. It also includes running directly an interactive IPython notebook (similar to Wakari). For this tutorial, I'll focus on running processes from a user's computer. Note that the PiCloud documentation contains several additional examples and tutorials for readers to get inspiration from, such as running applications on PiCloud (such as R), and even deploying Matlab-compiled functions.

Depending on the operating system, the "*Get Started*" guide on PiCloud's Web interface advises on how to install Python, the PiCloud library, and PiCloud credentials on the local system providing access to the PiCloud infrastructure. (Note that for a single account, many credentials can be generated—thus, one account can be used by many users. This facilitates logistics in a group/lab setting.)

For a Python script to have access to the PiCloud infrastructure, it must include

```
>>> import cloud
```

Now, let's define a simple Python function:

```
def add(x, y):
return x+y
```

Issuing the command

```
>>> jid = cloud.call(add, 1, 2)
1
```

will execute "`add`" on the cloud, and obtain a job id (`jid=1`), which we can use to monitor the process of the job(s):

```
>>> cloud.status(1)
```

The returned messages are self-explanatory; for example, "done" indicates that the process has completed.

The results of the job "add" can be seen with:

```
>>> cloud.result(1)
3
```

Thus, this simple example shows that PiCloud automatically transferred the dependencies of the function "add" to the cloud, and executed the process on the cloud.

One of the most useful capabilities of PiCloud is to map a function with different inputs—in other words, execute the same Python function with different arguments on the cloud (the way the Python `map` command does it on the local Python):

```
>>> jids = cloud.map(add, [1,3,2], [2,2,2])
```

This actually launches three jobs, the results of which can be obtained via their ids as

```
>>> cloud.result(jids)
3,5,4
```

Even this trivial example illustrates the power, but easiness and efficiency, of distributing several identical processes (with different inputs) on PiCloud, since PiCloud takes care of sending the function to the cloud and executing it with all possible arguments. According to the selection of instances, jobs will start as soon as "on-demand" cores are available (recall that PiCloud predicts usage by profiling the jobs of each user and assigns priorities), for which the user pays only for the amount of time used for computation (at the millisecond level). Alternatively, "real-time" cores guarantee the availability of concurrent cores. However, a setup wait time is necessary, and they do incur a minimum cost (their cost is the same as with on-demand cores if usage is greater than 36 minutes/hour). PiCloud offers several core types, varying in memory, computer power, local disk space, and price. For example,

```
>>> jids = cloud.map(add, [1,3,2], [2,2,2], _type='m1')
```

will map the "add" function on "m1"-type cores.

Now let's consider a problem for which we need to use an external binary or library. While the standard PiCloud instance contains Python and several libraries, it can be extended using environments. An environment is a Linux Container that runs on top of the standard PiCloud Linux Container (PiCloud refers to this as the *base*). As with Wakari and CPUsage, within environments the user can install applications necessary for computation. The environments can be private, shared among users, or public. The shared environment facilitates research in a group/lab setting, since many users can update/modify the environment keeping the installed codebase up to date. Public environments (as with public AMIs) are available to the broad community and anyone can use them, increasing the reproducibility of science, and the exchange of information.

PiCloud's documentation and blog contains several examples regarding calling external binary applications (for example, FFmpeg to transcode video), but not one together with how to create a custom environment. I'll use as an example a rather (scientifically) complex problem from medical image processing, and particularly neuroimaging. This will demonstrate the power of PiCloud and resonate with many scientists.

The scientific problem at hand is to have imaging volumes (3D matrices) overlap as much as possible, commonly known in neuroimaging as normalization. Usually, this involves a reference volume, known as the atlas, which is considered as the fixed volume (because it doesn't change), and a subject volume, which moves (that is, it's transformed) to match the atlas. The best matching is usually obtained via nonlinear registration between the 3D volumetric data,[5] to find a mapping between voxel locations of the subject to those of the atlas. Overall, this process is complex and some registration frameworks require iterative optimization of millions of parameters of the transformation. Depending on the volumes' size, the amount of deformation present (that is, how much do we need to change the subject to match the atlas), and the algorithm used, it can take several hours for a single pair of inputs. Registration is repeated for all input subjects; thus, processing needs increase rapidly.

For this demonstration, I'll use the Advanced Normalization Tools (ANTs) collection of applications for neuroimaging (http://sourceforge.net/projects/advants) version ANTS_1_9_y. The first step is to ensure that ANTs is available within our PiCloud environment. We follow PiCloud's documentation for the environment, to launch an ssh terminal inside the browser, which gives access to an instance running our environment. With the ssh terminal open, we follow the instructions (see http://brianavants.wordpress.com/2012/04/13/updated-ants-compile-instructions-april-12-2012) to download, compile, and install the ANTs toolkit (PiCloud gives root access with the sudo command). At the end, issuing the command

```
>/home/picloud/ants/bin/ANTS
```

will return the command usage of the ants binary, indicating its proper installation. We'll refer to this environment with the name "test."

The next step is to upload our data to PiCloud for processing (readers can download example data from www.oasis-brains.org or use plain 2D images with necessary modifications to the calls of the ANTS commands). For simplicity, we assume that the imaging data reside in a local "data" folder and are named sequentially s01.nii.gz to s10.nii.gz. (nii.gz is the common NIFTI imaging format; see http://nifti.nimh.nih.gov). The goal is to register them to an atlas, named atlas.nii.gz. In the Python interpreter (or with a `for` loop within a script), the commands

```
>>> import cloud
>>> cloud.bucket.put('data/atlas.nii.gz', 'atlas.
  nii.gz')
>>> cloud.bucket.put('data/s01.nii.gz', 's01.nii.gz')
…
>>> cloud.bucket.put('data/s10.nii.gz', 's10.nii.gz')
```

will upload the data on the PiCloud storage as buckets (a key-value interface to storing data objects following the definition of Amazon S3 storage buckets).

Now, let's define the Python script, shown in Listing 1 (see Figure 1), which performs a registration between the two volumes, atlas, and subject, and then deforms the subject volume to match the atlas.

This script consists of two parts: one defining the function `job`, and `main`. Here, `job` uses two binaries from ANTs: *ANTS* for registration and *WarpImageMultiTransform* to deform a volume after the registration has occurred. It first defines the atlas (atlas.nii.gz) and subject volumes (sXX.nii.gz), the filenames of which are stored in the strings `atlas` and `subject`, respectively. It then downloads from the cloud storage the files using the `cloud.bucket.get` PiCloud function. Subsequently, Python's `subprocess.Popen` invokes the *ANTS* binary to register the subject to the atlas, with several parameters defined to perform this nonlinear registration (for more information on the parameters see the ANTs manual; http://sourceforge.net/projects/advants/files/Documentation/ants.pdf). The `popen.wait()` command locks, waiting for the external (`Popen`) call to finish before continuing with the remaining commands. The outputs of *ANTS* are several large data files that define the nonlinear forward and backward deformation field (sXXWarp.nii.gz and sXXInverseWarp.nii.gz) and a text file defining the affine (linear) transformation parameters (sXXAffine.txt). Subsequently, using the *WarpImageMultiTransform*, the function takes the subject and actually deforms it to the space of the atlas volume to match it. The final outcome is the volume sXX_warped.nii.gz, which is uploaded and stored in the cloud with the `cloud.bucket.put` function.

At the end of the script, in its `main` part, there's a `cloud.map` invocation with several arguments. Here's the simplicity

```
import cloud
import subprocess
import time
import os
import sys

def job(n):
    """
    Performs a non-linear registration between atlas  'atlas.nii.gz' and a subject sXX.nii.gz.

    Input to job (e.g., for subject 1):
          - atlas.nii.gz
          - s01.nii.gz

    non-linear registration between moving image (e.g., s01) and fixed image atlas
    ANTS 3 -m CC[atlas.nii.gz,s01.nii.gz,1,2] -o s01 -i 120x120x120x120 -t SyN[0.25] -r
Gauss[3,0.] --affine-metric-type CC --number-of-affine-iterations
10000x10000x10000x10000x10000

    Output from job (e.g., for subject 1):
          - s01_warped.nii.gz

    Keyword arguments:
          n -- n-th subject id (moving image)
    """
    print 'Subject %d' % (n)
    subject = 's' + str(n).rjust(2, '0')

    # get input data from cloud storage
    indata = ['atlas.nii.gz', subject + '.nii.gz']
    for ff in indata:
          cloud.bucket.get(ff)

    # non-linear registration between moving image subject and fixed image atlas
    args = '/home/picloud/ants/bin/ANTS 3 -m CC[atlas.nii.gz,' + subject + '.nii.gz,1,2] -o '
+ subject + ' -i 120x120x120x120 -t SyN[0.25] -r Gauss[3,0.] --affine-metric-type CC --
number-of-affine-iterations 10000x10000x10000x10000x10000'
    print '\n' + args
    start = time.time()
    popen = subprocess.Popen(args, shell=True, stderr=subprocess.STDOUT)
    popen.wait()
    elapsed_ants = time.time() - start
    print 'Elapsed time: %.2f s' % (elapsed_ants)

    # warp (transform) the moving image subject to the fixed atlas image
    args = '/home/picloud/ants/bin/WarpImageMultiTransform 3 ' + subject + '.nii.gz ' +
subject + '_warped.nii.gz ' + subject + 'Warp.nii.gz + subject + 'Affine.txt -R atlas.nii.gz'
    print '\n' + args
    popen = subprocess.Popen(args, shell=True, stderr=subprocess.STDOUT)
    popen.wait()

    # upload output data to cloud storage
    outdata = [subject + '_warped.nii.gz']
    start = time.time()
    for ff in outdata:
          cloud.bucket.put(ff)
    elapsed_put = time.time() - start

if __name__ == '__main__':
    """
    perform non-linear registration between subjects sXX.nii.gz and an atlas.nii.gz
    """
    subjects = range(1,10)
    jids = cloud.map(job, subjects, _env='test', _type='f2', _label='cise_test')
    print '%d jobs, IDs = %d..%d' % (len(jids), jids[0], jids[-1])
```

Figure 1. Listing 1. A Python script that performs a registration between the two volumes, atlas, and subject, and then deforms the subject volume to match the atlas.

of PiCloud. With one command, the function `job` will be executed for all subjects s01 … s10, with the environment "test" on the "f2" core type. With on-demand cores, it's possible that the jobs are queued; however, if we wanted guaranteed simultaneous execution, we would reserve real-time cores (as many as the number of subjects) from the PiCloud dashboard. Note: the user should release the real-time cores after the end of computation, to ensure that charges don't continue. Overall, registration can take up to two hours (which explains the `time.time()` inclusions in the script). The user can check the processing times and explore the speedup offered by each core type. The ability to run all registrations at the same time (one on each core) demonstrates the acceleration achieved with this parallelization.

Once the computation has completed (which can be checked either on the PiCloud dashboard or via the `cloud.result` function), we can download the files to a local results folder with

```
>>> cloud.bucket.get('s01_warped.nii.gz', 'results/s01_warped.nii.gz')
```

that can be repeated for all other files. Cloud stored files can be deleted with the `cloud.bucket.remove(filename)` command to reduce storage costs.

The possibilities with PiCloud are endless, and it's truly a powerful platform. If the reader wants to use another binary installed in the environment, the aforementioned listing can be modified to suit many needs. With some knowledge of Python and Linux, PiCloud permits utilizing cloud instances with transparency and ease of use, when relying on default behavior. Furthermore, the PiCloud library has an abundance of functions and parameterizations to satisfy complex operations and cases. It provides several options to synchronize local and cloud volumes, which can be mounted to the file system, thus facilitating data transfer and workflow. Recall that the PiCloud is nonblocking, so commands on the cloud will continue to the remainder of the script. For this purpose, there exist also several options (such as the `cloud.join` function) for building and synchronizing complex workflows with queues (see relevant documentation).

Practical Considerations and Risks

With easily parallelizable tasks, processing on the cloud is worth considering and could be beneficial for a variety of users.[2] It could complement an in-house or in-campus computational infrastructure due to its elasticity (that is, the possibility to add instances on demand). Several commercial and open source tools aim to simplify this process. However, there exist several lessons to be learned, challenges to tackle, and opportunities to explore.

The characteristics of the computational problem and the amount and type of data that must be moved in, out, and around the cloud (from cloud storage to processing instances) should be considered carefully. Applications that require homogeneous and optimized environments with very long computational times and frequent utilization might not be ideal for the cloud.[3,4] Even embarrassingly parallel tasks that involve heavy data transfers need special attention. For example, in neuroimaging, some of the input data are in the tens of megabytes, but metadata produced can be in the hundreds of megabytes (in our example, the Warp and InverseWarp fields). This means that while uploading data might not be a problem, downloading the metadata and shuffling them around the cloud can be an issue. Thus, proper care in designing the workflow is necessary; otherwise, the benefit of large-scale computing on the cloud could be lost. Another issue is data privacy, and every user must value this aspect on its own merits.

Even though it isn't immediately apparent to most scientists, good quality code is important. Although previously, inefficient code (such as unnecessary loops or poor memory usage) would have just caused slow computation, now the same code will actually cost money. In the world of big data, when the same code is used thousands of times, even small inefficiencies do add up.

Another aspect contributing to cost is the use of commercial software that does require a paid license. Licensing on the cloud is not yet straightforward and requires communicating with the software's maker to identify possibly solutions. On the other hand, relying more on open source and free software reduces deployment costs—and Python, with its vast libraries, is a great start.

From a logistics viewpoint, using the cloud for research can be a challenge due to billing and accounting regulations. From a personal experience, getting reimbursed for cloud computing costs (on a personal credit card) from research funds was an effort not worth investigating in my current post in Europe, while in the USA this was more straightforward.

On the positive side, a great opportunity arises in educating students (and young researchers in general) to appreciate that although their thesis/research might not need HPC per se, (future) problems can benefit from large computational resources (either local academic clusters or the cloud). This experience can challenge students not only into writing better code and considering open source software, but also into rethinking their own problem (and maybe coming up with a cloud-suitable implementation). It's also a current need in the marketplace, so young researchers have much to gain. However, as of now, discouraging this may be the lack of simplicity, and that some familiarity with Python and Linux are required.

Many institutions around the world are beginning to adopt new perspectives of teaching computing to undergraduate and graduate students. New online courses, for example the High-Performance Scientific Computing course in Coursera (www.coursera.org/course/scicomp), also show this trend. In my current institution, in the first semester we offer graduate courses on Python, working with data, cloud

computing, and practical hands-on experience with such tools. In my Machine Learning and Pattern Recognition course, I emphasize algorithms that are easy to parallelize (in a mapReduce-like fashion). I also teach a course on how to design image analysis and processing algorithms tailored to large dataset sizes and the cloud.

Clearly, the cloud can provide interesting alternatives, supplementing local computational infrastructures. However, this additional option might confuse the user even further. He or she must still make the—sometimes complex—decision on where to perform the computation (via a local desktop, local cluster, private, or public cloud), taking into account cost and computational complexity. The management of such hybrid setups and the resource allocation particularly from the provider's side are extremely active topics. However, currently no transparent and easy-to-use approach is available for the user. In the long run, such a mixed environment will permit an automated, and seamless to the user, delegation of where tasks are executed. The user, for example, will still work on his local Python installation, but behind the scenes according to the profiling of the jobs/tasks, decisions will be made, automatically, to push some of the tasks to local clusters (or private clouds) or public clouds. This implies superior profiling of the tasks, which is possible for heterogeneous tasks (for example, via learning from historical information of several execution variables, or via source-code profiling). In other domains, such as in medical image processing, where many tasks are homogeneous but execution time/needs depend on data content and context, until recently profiling wasn't possible. We recently showed, that we can extract privacy-preserving features from the imaging data and learn associations with execution time/resources;[5] thus, resource allocation for such data-dependent tasks can be improved.

I would like to conclude this article with a warning. As with proprietary closed source software, there's a risk of vendor lock-in: the user can get "trapped" into relying on a single platform for computational needs, and changing platforms (out of choice or need) might require a significant redesign of the computational pipeline. Even more, when relying on early stage commercial platforms, there's the risk of product discontinuation. For example, PiCloud recently was acquired by Dropbox (www.dropbox.com), which will seize support and development for PiCloud. However, in consideration of its academic clients, PiCloud will be sustained as an open source tool, and the platform will be maintained by another entity (Multyvac). Even then, such transitions don't always guarantee that everything will remain in place: for example, Multyvac is considering not permitting function publishing via REST APIs (due to lack of user interest). Overall, openness and contingency plans are necessary if start-ups are to convince users to adopt their platform, and they should be communicated clearly and early on to the user. Thus, careful planning to eliminate as many risks as possible is required. Researchers comfortable with Linux and Python should consider open source tools, such as StarCluster and now PiCloud, and deploy their own solution. They should consult their IT office and explore solutions around these options. On the other hand, if researchers don't have such knowhow, it's best they contact one of the commercial providers, discuss their computational needs, and obtain assurances about the platform's continuity and viability. ■

Acknowledgments

I would like to thank Ken Elkabany, CEO/founder of PiCloud, Antony Tin, founder of Multyvac, Jeff Martens and Matt Wallington, CPUsage's co-founders, and Massimo Minervini and Rafael Uriarte, both PhD candidates at the IMT Institute for Advanced Studies, Lucca, for useful discussions. I would also like to acknowledge the support of a Marie Curie Action "Reintegration Grant" (no. 256534) of the European Union's Seventh Framework Programme (FP7).

References

1. T.E. Oliphant, "Python for Scientific Computing," *Computing in Science & Eng.*, vol. 9, no. 3, 2007, pp. 10–20.
2. A. Gupta and D. Milojicic, "Evaluation of HPC Applications on Cloud," *Proc. 2011 Sixth Open Cirrus Summit*, 2011, pp. 22–26.
3. T. Dillon, C. Wu, and E. Chang, "Cloud Computing: Issues and Challenges," *Proc. 2010 24th IEEE Int'l Conf. Advanced Information Networking and Applications*, 2010, pp. 27–33.
4. A.G. Carlyle, S.L. Harrell, and P.M Smith, "Cost-Effective HPC: The Community or the Cloud?" *Proc. 2010 IEEE 2nd Int'l Conf. Cloud Computing Technology and Science*, 2010, pp. 169–176.
5. M. Minervini et al., "Large-Scale Analysis of Neuroimaging Data on Commercial Clouds with Content-Aware Resource Allocation Strategies," *Int'l J. High-Performance Computing Applications*, 2014, preprint; http://hpc.sagepub.com/content/early/2014/01/15/1094342013519483.abstract.

Sotirios A. Tsaftaris is with the IMT Institute for Advanced Studies, Lucca, Italy, and Northwestern University. His research interests are medical image analysis, computational biology, machine learning, and large-scale analysis of imaging data. Tsaftaris has a PhD in electrical and computer engineering from Northwestern University. Contact him at s.tsaftaris@imtlucca.it.